1982

Optimistic concurrency control for distributed databases.

McElyea, William Peyton.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

OPTIMISTIC CONCURRENCY CONTROL FOR
DISTRIBUTED DATABASES

by

William Peyton McElyea

June, 1982

Thesis Advisor: D. Z. Badal

Approved for public release, distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Optimistic Concurrency Control for Distributed Databases	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis June, 1982
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): William P. McElyea	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	12. REPORT DATE: June, 1982
	13. NUMBER OF PAGES: 68
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Optimistic
Concurrency Control
Distributed Database

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

One of the most important considerations in developing a distributed database system is the concurrency control mechanism. Recently, many arguments have been advanced in favor of the optimistic solution to concurrency control. This work reviews two algorithms that apply the Kung-Robinson proposal to a distributed database system. A different algorithm originally proposed by Badal is developed and expanded. This new algorithm switches from an optimistic mode of detecting and resolving non-serializable execution to a

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (continued)

pessimistic mode of preventing non-serializable execution when the degree of conflict reaches a certain level. In other words, the algorithm adapts itself to the degree of conflict. Representative optimistic algorithms are then compared with two-phase locking and two-phase commit under different scenarios. Conclusions are drawn based on the performance of the algorithms under the different scenarios. The new algorithm appears to perform better than any of the other concurrency control mechanisms.

Approved for public release, distribution unlimited

Optimistic Concurrency Control for Distributed Databases

by

William P. McElyea
Major, United States Marine Corps
B.A., Brown University, 1968

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1982

ABSTRACT

One of the most important considerations in developing a distributed database system is the concurrency control mechanism. Recently, many arguments have been advanced in favor of the optimistic solution to concurrency control. This work reviews two algorithms that apply the Kung-Robinson proposal to a distributed database system. A different algorithm originally proposed by Badal is developed and expanded. This new algorithm switches from an optimistic mode of detecting and resolving non-serializable execution to a pessimistic mode of preventing non-serializable execution when the degree of conflict reaches a certain level. In other words, the algorithm adapts itself to the degree of conflict. Representative optimistic algorithms are then compared with two-phase locking and two-phase commit under different scenarios. Conclusions are drawn based on the performance of the algorithms under the different scenarios. The new algorithm appears to perform better than any of the other concurrency control mechanisms.

TABLE OF CONTENTS

I. INTRODUCTION

The desirable characteristics of a distributed database system are well known. The roots of these characteristics are found in the concepts of locality, availability, reliability and concurrency. The ability of such systems to use small, local computers rather than large mainframes is an important economic consideration. The 1980's may well be the decade of distributed computing systems.

One of the most important considerations in developing a distributed database system is concurrency control. Some means is needed to ensure that each transaction executes on a consistent database. A database is a set of data objects, which are related in certain ways. If these relationships are viewed as assertions about the data objects, then a consistent database is one that satisfies all of its assertions. [Ref. 1] A transaction is a unit of consistency, which moves the database from one consistent state to another consistent state. Since a transaction is usually made up of several atomic actions and since all of these atomic actions cannot all be completed at exactly the same instant, it is often necessary for a database to become temporarily inconsistent when moving from one consistent state to another consistent state. The concurrency control mechanism must either prevent transactions from executing on the inconsistent portions of the database or prevent transactions from committing if they have accessed inconsistent portions of the database. Serial equivalence is used to ensure that a transaction

leaves the database in a consistent state. If all transactions execute serially, that is, one at a time, then each transaction leaves the database in a consistent state. In other words, no inconsistencies are introduced since each transaction completes its execution before the next transaction begins its execution. The concurrent execution of several transactions, therefore, is correct only if it produces the equivalent results that could be obtained by running the same transactions in serial order. [Ref. 2] Such execution is said to be serializable. Execution, which cannot be duplicated by a serial execution of transactions, is said to be non-serializable and is not allowed.

There are three basic strategies for ensuring that a transaction always executes on a consistent database. The first strategy, which is based on locking, requires that a transaction acquire an exclusive lock on each data object that it wishes to update and a shared lock on each data object that it wishes to read. The locks are usually acquired on demand during run time. In order to facilitate easy recovery, the locks are usually held by the transaction until commit.

The second strategy assigns each transaction a time-stamp, which is unique throughout the system. Transactions are required to execute in the order of their time-stamps.

The third strategy, which is usually called optimistic, allows each transaction to execute freely without acquiring locks and without regard to a specific ordering. At the end of transaction execution, the objects read and updated by the transaction are reviewed to ensure that the transaction viewed the database in a consistent state. If so, the transaction commits. If not, the transaction is aborted and restarted.

Recently many arguments have been advanced in favor of the optimistic solution to concurrency control. All of the arguments assume a low probability of transaction conflict. It is often argued that under optimistic concurrency control, the overhead associated with nonconflicting transaction synchronization is low. There is a potentially high synchronization overhead for conflicting transactions and a potentially high overhead during recovery. On the other hand, if the probability of conflict is high, then the synchronization overhead associated with nonconflicting transactions, which are synchronized by locks or time-stamps, might be justified. If the probability of conflict is high, then the conflicting transaction overhead under an optimistic concurrency control mechanism will become dominant.

In many real-life applications the probability of conflict is low. In recent years, therefore, several proposals for optimistic concurrency control have been suggested. Kung and Robinson [Ref. 3] developed a method which is appropriate for a centralized database. Ceri and Owicki [Ref. 4] expanded this approach to apply to a distributed database. Schlageter [Ref. 5] also developed an algorithm for a distributed system based on the ideas of Kung and Robinson. Badal [Ref. 6] suggested a different approach in an algorithm that was originally designed for a distributed database. The purpose of this paper is to examine and compare the work that has been done so far in the area of optimistic concurrency control for a distributed database. Chapter 2 will summarize the algorithms, which are based on the Kung-Robinson proposal. Chapter 3 will develop the algorithm originally suggested by Badal. Chapter 4

will compare the performance of the optimistic concurrency control mechanisms with two-phase locking. Chapter 5 will present conclusions.

II. ALGORITHMS BASED ON THE KUNG-ROBINSON PROPOSAL

This chapter gives a brief description of the Kung-Robinson proposal and the distributed algorithms which are based on this proposal.

A. KUNG and ROBINSON [Ref. 3]

This algorithm divides a transaction into three phases, a read phase, a validation phase and a write phase. During the read phase local copies of all data objects to be updated are created and all writes are directed to these copies. The original data objects continue to be available for other transactions. The validation phase determines whether or not the transaction will cause a loss of integrity for the data base. A transaction will cause a loss of integrity if the transaction leaves the database in an inconsistent state. If the transaction will cause a loss of integrity, then the transaction is aborted and restarted. If the transaction successfully validates, then it enters a write phase and the local copies are substituted for the original data objects.

The concurrency control mechanism maintains sets of object names, which are accessed by a transaction. These sets, a read set and a write set for each transaction, are initialized to be empty. An addition is made to the read set whenever a transaction reads a value or pointer from a node. Returning a value from a query is treated as a write. Thus, whenever a transaction wants to change a date object or return the value of an object, an addition is made to the write set.

Serial equivalence is used to ensure that a transaction leaves the database in a consistent state. Each transaction, T(i), is assigned a unique transaction number t(i). The following conditions are used during the validation phase to establish serial equivalence of transaction T(j) with transaction number t(j) and for all T(i) with $t(i) < t(j)$:

(1) T(i) completes its write phase before T(j) starts its read phase. In other words, T(i) completes before T(j) starts.

(2) The write set of T(i) does not intersect the read set of T(j), and T(i) completes its write phase before T(j) starts its write phase. In other words, the writes of T(i) do not affect the read phase of T(j) and T(i) finishes writing before T(j) starts writing.

(3) The write set of T(i) does not intersect the read set or the write set of T(j) and T(i) completes its read phase before T(j) completes its read phase. In other words, T(i) does not affect the read phase or the write phase of T(j).

Serial validation implements conditions (1) and (2). Condition (2) requires that the write phases must be serial. This is done by placing the assignment of a transaction number, validation and the subsequent write phase all in a critical section. The critical section write locks the entire data base. This method is then optimized by moving as much of the work as possible outside of the critical section. This shortens the length of the critical section and allows a greater degree of parallelism.

Parallel validation implements all three conditions. Conditions (1) and (2) are checked for transactions which have completed their write

phase. Condition (3) is checked for transactions, which have completed their read phase but have not yet completed their write phase. An Active Set List (ASL) is maintained for transactions, which have not yet completed their write phase. When a transaction finishes its read phase, it enters a short critical section in which it copies the ASL and adds itself to that list. The transaction then validates by ensuring that its read and write sets do not intersect the write set of any transaction in the ASL. When a transaction commits or aborts, it removes itself from the ASL.

A possible problem with this algorithm exists if a transaction repeatedly fails validation. The transaction will be continuously restarted and will never run to completion. This is called starvation. A solution to this problem is offered by Kung and Robinson. It is suggested that the number of times a transaction is forced to restart be recorded and if this number exceeds a certain limit, then the transaction should remain in a critical section and be allowed to run to completion.

B. CERI and OWICKI [Ref. 4]

This algorithm extends the Kung-Robinson proposal to a distributed data base. It ensures consistency by forcing each transaction to execute in the same relative order at each node. A transaction originates at a node, which is then the master site for that transaction. The transaction master initiates subtransactions, which are logically part of the transaction, but which execute at different nodes. Each subtransaction performs its read phase and is validated at its local node using the parallel validation procedure of Kung and Robinson. If this

validation fails, then the subtransaction is backed up and restarted locally. If local validation succeeds, then the subtransaction enters its global validation phase.

The global validation phase consists of constructing a "happened before" set (HB set) and ensuring that each transaction in this set either commits or aborts. The HB set is constructed in the following manner. Originally the HB set consists only of the subtransaction which is being validated. Then the transitive closure of this set and the copy of the Active Set List (ASL), which was used in the local parallel validation, is constructed. Each transaction in the ASL is compared to every member of the HB set. If the transaction is in conflict' relation with any member of the HB set, it is added to the HB set. A transaction T(i) is in conflict' relation with transaction T(j) if the following statement evaluates to true:

$$\left[(RS(i) \cap WS(j) \neq \Phi) \vee (WS(i) \cap RS(j) \neq \Phi) \vee (WS(i) \cap WS(j) \neq \Phi)\right] \wedge T(i) < T(j)$$

where RS(k) is the read set of T(k) and WS(k) is the write set of T(k). This procedure is continued until no more transactions can be added to the HB set. The subtransaction, which is being validated, and all local transactions are then eliminated from the HB set. (Local transactions are included in the construction of the HB set because two global transactions can conflict due to the effects of local transactions.) The final version of the HB set contains only global transactions which may be in conflict with the subtransaction, which is attempting to validate. Global validation consists of verifying that all of the transactions in

the HB set have either committed or aborted. This information can be read from the Active Set List. (Remember, the HB set was constructed using a copy of the ASL.) If all of the members of the HB set have either committed or aborted, then the subtransaction sends a message to the master site saying that it is ready to commit. If there are still active members of the HB set, then the subtransaction waits for a certain "time out" period and again checks to see if there are any active members of the HB set. If not, the subtransaction sends the "ready to commit" message. If there are still active members, the subtransaction removes itself from the ASL and sends an "abort" message to the master site. The master site oversees a two-phase commit. If none of the subtransactions abort, then the master site sends a commit message to all subtransaction sites. If any of the subtransactions abort, then the master sends an abort message to all subtransaction sites. After receiving the commit or abort message from the master site, each subtransaction enters a critical section, is assigned a transaction number, if the message is to commit, and updates its status in the ASL.

C. SCHLAGETER [Ref. 5]

This algorithm modifies the Kung and Robinson proposal and then extends the modified method to the distributed case. There is a separate protocol for global update transactions and for global read transactions. The protocol for global update transactions guarantees consistency by ensuring that all subtransactions are in a "tentatively written" phase at the same time. The protocol for global read transactions guarantees consistency by ensuring that all subtransactions are in the read phase at the same time.

The Kung and Robinson proposal is modified so that local read-only transactions are freed from any consideration of concurrency control. (This method has to be modified in the case of a distributed system.) There is no validation phase for a read-only transaction. A read set, which indicates all objects that are read by each transaction, is maintained. An update transaction considers the read sets of parallel read transactions and the read and write sets of other update transactions. If conflict is detected with a read-only transaction, then the update transaction is deferred until after the completion of the conflicting read transaction. After the completion of the read transaction the validation of the update transaction must be repeated to detect read-only transactions, which began after the original validation and which are still in progress. Conflict with an update transaction results in an abort and restart.

The algorithm is applied to the distributed case in the following manner. A non-redundant distribution of data is assumed. A global transaction is executed as a set of local subtransactions. The subtransactions are coordinated by a primary subtransaction. A two-phase commit protocol guarantees the atomicity of global transactions.

Global update transactions are validated against local and global update transactions. When a subtransaction completes its read phase it enters a validation phase. A global update subtransaction must validate as described in Kung and Robinson and must also validate against other global update subtransactions at its site, which have tentatively written but not finally written. If validation is successful, a transaction

number is assigned and a tentative write is done. Since a transaction number is assigned at tentative write, this write is seen as a normal write as far as validation is concerned. In the tentative write phase either global write or backup is possible.

In order to ensure that a transaction always executes on a consistent database, it is necessary to prevent any transaction from modifying data, which is tentatively written but not finally written, and to prevent any transaction from validating if it has read tentatively written objects. This means that both local and global read transactions and local update transactions must also validate against the set of transactions which have tentatively written but not finally written.

An update subtransaction sends a message to the primary subtransaction either when it aborts or when it enters the tentative write phase. If all subtransactions are tentatively written then the primary subtransaction sends a message to all of the subtransactions telling them to make the results of their tentative write global, that is, to do a final write. If any subtransactions abort, then the primary subtransaction sends an abort message to all subtransactions and the transaction is restarted.

A global read transaction also utilizes a primary subtransaction. When each subtransaction begins its read phase it sends a "started" message to the primary subtransaction. Each subtransaction must remain in the read phase at least until a "validation allowed" message is received from the primary subtransaction. The "validation allowed" message is sent after the primary subtransaction receives a started

message from each subtransaction. After the read phase, the subtransaction validates against global update subtransactions at its site. If validation is successful, a subtransaction sends an "OK" message to the primary subtransaction along with the results of the read if required. If the primary subtransaction receives an "OK" message from each subtransaction, then the transaction is successfully terminated. If an abort message is received from any subtransaction, then the transaction is restarted.

III. DEVELOPMENT OF BADAL ALGORITHM

This chapter develops the algorithm originally proposed by Badal [Ref. 6]. The first section introduces the transaction model. The second section presents definitions and concepts that are needed to understand the algorithm. The third section presents an overview of a transaction; it includes a detailed example to illustrate the workings of the algorithm.

A. TRANSACTION MODEL

In our model, a transaction is seen as a set of atomic actions. Each atomic action is either a read or an update on a single data object. If the action is an update, then a temporary version of the data object is produced and made visible to other transactions. Atomic actions, whose results are used by subsequent atomic actions or which are dependent on the results of a previous atomic action, are grouped together into a subtransaction. An atomic action, which is independent of all other atomic actions, is also a subtransaction. Thus, while a transaction is physically a set of atomic actions, it is logically a set of subtransactions. The subtransactions of a transaction can execute concurrently or sequentially. Since subtransactions are groups of atomic actions, which are in some way dependent on the results obtained by other atomic actions within the subtransaction, the execution of the atomic actions within a subtransaction must be sequential. The transaction as a whole is also an atomic action in the sense that either the entire transaction is completed and committed or the entire transaction is aborted.

An example will clarify these ideas. T1 is a transaction.

```
T1:  read (a)
     if (a) < 2 then
         update (b)
     else
         update (c)
     update (d)
```

T1 consists of the following atomic actions:

```
read (a)

update (b)

update (c)

update (d)
```

Logically T1 consists of two subtransactions. One subtransaction, ST11, consists of reading the data object (a) and, depending on the value of (a), updating either (b) or (c). Thus, the atomic actions within the subtransaction ST11 must be done sequentially. The second subtransaction, ST12, consists of updating the data object (d). ST11 and ST12 can either be done concurrently or sequentially. T1 is atomic in the sense that either ST11 and ST12 must both commit or both abort.

In this model a transaction must enter and exit from the system at the same site. This site is called the initiating site. Once the transaction has entered the system, it can either move from site to site allowing its subtransactions to execute in a sequential manner or it can FORK and allow the subtransactions to execute concurrently. If a transaction FORKS, then the subtransactions JOIN at some predetermined site once they have completed their execution. The site of the JOIN is usually either the site of the FORK or the initiating site. A transaction attempts to detect and resolve non-serializable execution as it

moves from site to site. When a transaction completes its work and returns to its initiating site, it must ensure that it has generated only serializable execution. Prior to exiting from the system, a transaction must either commit its temporary versions or it must abort.

B. DEFINITIONS AND CONCEPTS

1. DO Log

Each named data object in the data base has associated with it a DO log. The DO log operates as a stack. Each transaction, which reads or updates data at an object, pushes an entry onto the data object's DO log. An entry consists of the transaction identification, a file or a list of fields and records read or updated, the status of the transaction (temporary, committed, aborted), the previous conflict history of the transaction, and a metric reflecting the amount of work done so far by the transaction.

The DO log is written to stable storage as part of the atomic action on the data object. This does not impose any additional burden on the system since this information must be kept anyway for recovery purposes. When a data object is moved to main memory, a copy of the entries in its DO log, which are marked as temporary (i.e., the transaction which made the entry has not committed or aborted), are moved with it.

2. Granularity

The finest possible granularity can be used since the DO log entry can specify the exact record and fields that are read or updated.

3. Priority of Transactions

It is assumed that all transactions have the same priority.

4. Conflict

A transaction, T2, is in conflict with transaction T1 if T2 reads any of the same fields or records which were updated by T1 or if T2 updates any of the same fields or records, which were read or updated by T1, and if T1's DO log entry is still marked as temporary. In other words, if T2 reads after T1 reads, it is not considered to be a conflict. The conflict occurs only if a transaction read is followed by another transaction writing the same data object or if a transaction update is followed by another transaction reading or writing the same data object and the first transaction has not committed.

5. Conflict History

Each transaction has a conflict history. When a transaction enters the system, its conflict history is empty. An entry is made in the conflict history whenever the transaction is in conflict with another transaction at a particular data object. Each entry consists of the identification of the transaction with which the conflict occurs, the data object at which the conflict occurs and a metric to reflect the amount of work involved in the atomic actions performed by the conflicting transactions. The current conflict history is deposited at each data object accessed by the transaction. It is deposited as part of the DO log entry. When a transaction returns to its initiating site after it finishes executing, it sends a copy of its conflict history to the initiating site of each transaction in its conflict history.

6. Precedence Relations

Using its own conflict history and the conflict histories which are sent to it, a transaction constructs a precedence relation. The precedence relation shows a transaction's view of the order in which transactions executed at sites at which conflicts occurred. A precedence relation shows non-serializable execution has occurred if a transaction appears in more than one place in the relation.

The following example shows how a precedence relation is constructed and how the relation can show non-serializable execution. If transaction T1 conflicts with transaction T2 at data object c, this conflict would be represented in T1's conflict history as (T2T1 : c). If T1 receives (T1T3 : a) from transaction T3, which shows that T3 conflicted with T1 at data object a, then T1 would construct the following precedence relation:

T2 T1 : c

T1 T3 : a

This precedence relation shows that from T1's point of view the transactions executed in the order T2, T1 and finally T3. Thus, when a transaction receives information from other transactions in the form of conflict histories, the transaction is able to add information to its own conflict history. In this case, T1 adds T3 to its conflict history. If the resulting precedence relation does not indicate non-serializable execution, then the transaction sends a copy of the newly constructed relation to the initiating site of the transaction that it added to the relation. In this example, T1 sends the relation to T3. T3 uses the

relation to add information to the precedence relation that it has been constructing. Now suppose that T3 has constructed the following precedence relation:

T1 T3 : a

T3 T2 : b

When T3 adds T1's precedence relation to its own relation, it becomes obvious that non-serializable execution has occurred. T2 appears in the precedence relation in more than one place.

T2 T1 : c

T1 T3 : a

T3 T2 : b

That is, the precedence relation shows that no serial ordering of transactions could possibly have T2 execute before and after transactions T1 and T3.

The process of constructing precedence relations and sending the relation to the initiating site of the transaction added to the relation continues until non-serializable execution is detected and resolved or until it is established that only serializable execution has occurred. The fact that only serializable execution has occurred can be established when one of the transactions involved in the precedence relation commits or when no new transactions can be added to the relation.

If a precedence relation shows non-serializable execution has occurred, then serializable execution must be restored. When a precedence relation indicates that non-serializable execution has taken place, there should be some criteria for choosing which transaction pair will be chosen

to restore serializable execution. A cycle of non-serializable execution can be broken by re-executing one of the pairs of transactions in the reverse order. For example, if transaction pair T2T1 were re-executed at data object c in the order T1T2, the cycle would be broken. For the purpose of choosing the most economical manner of restoring serializable execution, a metric will be included with each transaction pair to indicate the relative cost of re-executing the pair. For example, the pair (T2T1 : c : 15) indicates that transaction T1 conflicted with transaction T2 at data object c and the relative cost for redoing the transaction pair is 15.

7. Temporary Versions

When a transaction accesses a data object either for a read or for an update, the transaction makes a DO log entry at the data object. This entry is marked as either temporary, committed or aborted. If a transaction conflicts with a previous transaction, which is still marked as temporary, then the DO log for the conflicting transaction is marked as t(w). This means that the conflicting temporary version is waiting for the previous transaction to either commit or abort. If the previous transaction aborts, then any transactions, which were based on its temporary versions, must also abort. If the previous transaction commits, then the next transaction in the DO log can change its entry from t(w) to t(r) and send a message to its initiating site saying that it is ready to commit. This means that now the temporary version of the data object is not based on a previous temporary version and, therefore, it is not waiting on the preceding temporary version to commit. It is ready to

commit. If a transaction does not conflict at a data object, then it can immediately mark its DO log entry as t(r).

8. Atomic Action Performed by a Transaction at a Data Object

The current transaction acquires a local lock on the data object. If the DO log contains any temporary versions with which the transaction will conflict, then the transaction must check for possible non-serializable execution. This is done in the following manner. If the DO log shows that a previous transaction has conflicted with the current transaction and if the current transaction is now conflicting with the previous transaction, then non-serializable execution will be generated.

For example, consider the case of two transactions, T1 and T2, which execute at three data objects, (a), (b), and (c). T1 arrives at (a) first and does an update. T2 arrives at (a) and sees that its update conflicts with T1. Since T1 has left an empty conflict history, however, the execution is still serializable. T2 will be updating the updated version left by T1. In the meantime, T1 has updated at object (b). T2 arrives at (b) and again detects a conflict. The execution is still serializable since T2 is again getting an updated version left by T1. Now T2 proceeds to (c) and performs an update. T2 does not detect any conflict and merely leaves its current conflict history at (c). T1 arrives at (c) and detects a conflict with T2. From an examination of T2's conflict history in the DO log, T1 can determine that there is no way to generate serializable execution since T2 has conflicted with T1 previously. Therefore, T1 sends a message to T2 telling it to "rollback" to (c) so that serializable execution can be restored.

If a transaction detects a conflict with a previous transaction but not non-serializable execution, then the transaction merely adds the previous transaction to its conflict history.

The transaction reads or updates the data object and pushes its entry onto the DO log. The transaction releases the local lock. If a new, temporary version has been produced, then it is immediately available to other transactions.

9. Locks

The most obvious problem with this optimistic concurrency control mechanism can be described as the "domino effect." A transaction after establishing that it did not generate non-serializable execution, makes its temporary versions immediately available to other transactions. If the transaction commits then this policy creates no problems. If one transaction aborts, however, then all other transactions, which have conflicted with the aborted transaction, must also abort. The domino effect does not represent a problem for applications in which transactions conflict rarely. It seems that most present applications are of this nature. In order to make this algorithm applicable to a wider range of situations, however, it would be desirable to eliminate or to restrict the domino effect. This can be achieved by extending the duration of the locks, which are held by transactions, in the following manner. As long as a transaction is not conflicting with other transactions it will execute in an optimistic mode, detecting and resolving non-serializable execution. It will acquire a local lock at a data object, execute at the object, release the lock and continue on to the next data object. If a

transaction conflicts with another transaction, however, the algorithm will automatically switch to a pessimistic mode of preventing non-serializable execution. That is, if a transaction conflicts with a previous transaction, which is still marked as temporary, then the conflicting transaction will not release the local lock when it finishes executing at the data object. The lock will be held until the previous transaction either commits or aborts. Of course, no other transactions are permitted to access these sequestered resources. Thus, the potential adverse impact of the domino effect can be limited by preventing any further access to the suspected data objects.

C. OVERVIEW OF TRANSACTION EXECUTION

First let's assume that no locks are encountered during transaction execution. Once the workings of the algorithm are detailed in this context, then the algorithm can be expanded to include the retention of locks and the resulting potential for deadlock. A transaction enters the system at its initiating site. The transaction consists of one or more subtransactions. The transaction can either execute its subtransactions sequentially or it can FORK and execute its subtransactions concurrently. The transaction carries its conflict history with it as it proceeds from site to site. If the transaction FORKS to execute subtransactions concurrently, then each subtransaction carries a copy of the transaction conflict history with it. The transaction or subtransaction attempts to detect and resolve non-serializable execution as it proceeds. The conflict history is updated and deposited at each data object. When a transaction completes its work, it returns to its

initiating site. Subtransactions, which are executing concurrently, JOIN at some predetermined site, either the site of the FORK or the initiating site. After the JOIN, the conflict histories of each subtransaction are merged into a transaction conflict history.

Prior to exiting from the system a transaction must ensure that it has generated only serializable execution and must commit its temporary versions. In order to detect non-serializable execution, a transaction must send a copy of its conflict history to the initiating sites of all transactions, which are listed in its conflict history. A transaction includes with its conflict history a metric, which measures the amount of work that would be involved in redoing the conflicting atomic actions.

If a transaction does not have any entries in its conflict history, then it is ready to commit. If such a transaction receives conflict histories from other transactions, then it sends messages to the initiating sites of the other transactions saying that it is ready to commit and that no non-serializable execution could have occurred between them. If a transaction does have entries in its conflict history and if it receives conflict histories from other transactions, then it constructs a precedence relation from all of the conflict histories. If the precedence relation indicates non-serializable execution then serializable execution is restored in the most economical manner. If the precedence relation does not indicate non-serializable execution, then the transaction sends a copy of the precedence relation to the transaction which it added to the relation.

The following is a summary of the rules, which must be followed in regard to conflict histories and precedence relations:

(1) If a transaction returns to its initiating site with an empty conflict history, it is ready to commit.

(2) If a transaction, which is ready to commit, receives a conflict history from another transaction, it sends a message to the initiating site of the conflicting transaction stating that it is ready to commit and, therefore, no nonserializable execution could have occurred between them.

(3) If a transaction returns to its initiating site with a nonempty conflict history, it sends a copy of its conflict history to the initiating sites of all transactions in its conflict history.

(4) If a transaction has a conflict history and receives conflict histories from other transactions, then it constructs a precedence relation from these conflict histories.

(5) If the precedence relation indicates nonserializable execution, then serializable execution is restored in the most economical manner.

(6) If the precedence relation does not indicate nonserializable execution, the transaction sends a copy of the precedence relation to the initiating site of the transaction that it added to the relation. (If more than one transaction is added to the precedence relation at the same time, then the copy is sent to the innermost transaction that was added. This situation occurs only when six or more transactions are involved in a cycle.)

Once a transaction has detected and resolved nonserializable execution, it is ready to enter its commit phase. A transaction can commit when all of the temporary versions, on which it based its temporary versions, commit. Each transaction leaves behind, as a part of its DO log, a record of the status of its DO log entry. This status can be either temporary, committed or aborted. A temporary DO version will be marked as either t(r), which means that it is a temporary version which is ready to be committed, or t(w), which means that it is a temporary version waiting for the temporary version on which it is based to commit or abort. A t(r) version is created whenever a transaction updates a committed version. A t(w) version is created whenever a transaction reads or updates a temporary version produced by another transaction. A t(w) version waits for the temporary version on which it is based to either commit or abort. It then sends a message to its initiating site saying either that it is ready to commit or that it had to abort. If a transaction receives a message saying that one of its temporary versions had to abort, then the transaction must rollback to the site of the abort and re-execute the remainder of the subtransaction. This, of course, may require conflict histories to be exchanged and non-serializable execution to be resolved again. Then the commit process continues with the t(w) versions sending messages to the initiating site when the temporary versions, on which they are based, commit. When all t(w) versions have acknowledged that they are ready to commit, then the transaction broadcasts a commit message to all sites at which the transaction executed. Upon receipt of these messages, the temporary versions are committed,

that is, they are made permanent and marked as such. Thus, the number of messages required for a transaction to commit is dependent on the number of conflicts that the transaction detected. One message is required from each site at which conflict occurred. One commit/abort message is also required for each site at which the transaction executed. This message tells the site to mark its DO log entry as committed or aborted.

Now let's examine what happens when locks are used in order to prevent a possible domino effect among conflicting transactions. A lock will be held whenever a transaction detects that it is in conflict with a transaction, which generated a temporary version, at a particular data object. When a transaction is executing and encounters a lock there are two possible outcomes. First, if the transaction waits, the lock may be released as the preceding transaction in the DO log commits. Second, the transaction holding the lock may be involved in deadlock. If this is the case, then the transaction encountering the lock may also be involved in deadlock. Thus, when a transaction encounters a lock, it waits for a certain period of time. If the lock is still present after the time-out period then the transaction sends a copy of its conflict history back to its initiating site. The initiating site takes the same actions that it would if the transaction had completed and returned to the initiating site. The transaction is marked, however, as still executing. Conflict histories are exchanged, precedence relations are constructed and non-serializable execution, if any, is detected and resolved. The only rule which needs to be modified slightly is the method for restoring serializable

execution. When locks are not used, serializable execution is restored in the most economical manner. When locks are used serializable execution should be restored in a manner which allows a transaction, which is still executing, to execute last. The reason for this is that if a conflict is not resolved in this manner, then when the transaction, which has not yet completed executing, continues after non-serializable execution has been resolved, it may create more non-serializable execution. This would occur if it again accesses a data object that was visited by another transaction which had been involved in the original non-serializable execution. If the still executing transaction is allowed to execute last, this will not be a problem. If more than one transaction is still executing, then it will not be possible to determine an exact ordering, which will preclude all future non-serializable execution. Thus, when more than one transaction is still executing, serializable execution is restored by selecting the most economical transaction pair, which will allow a still executing transaction to execute last.

The following algorithm is based on the preceding discussion. It divides the execution of a transaction into three phases, an execution phase, a detection phase and a commit phase.

```
EXECUTION PHASE:
FOR EACH SUBTRANSACTION DO CONCURRENTLY
OR SEQUENTIALLY
    FOR EACH ATOMIC ACTION DO SEQUENTIALLY
         CHECK FOR LOCK
         IF LOCK THEN
              WAIT FOR TIME-OUT
         IF NO LOCK THEN
              ACQUIRE LOCK
              INSPECT DO LOG FOR CONFLICT
              IF CONFLICT INSPECT DO LOG FOR NON-SERIALIZABLE
              (NON-SR) EXECUTION
                   IF NON-SR EXECUTION THEN
                        SEND MESSAGE TELLING CONFLICTING
                        TRANSACTION TO ROLLBACK TO SITE
                        OF NON-SR EXECUTION TO RESTORE
                        SERIALIZABLE EXECUTION IN MOST
                        ECONOMICAL MANNER
                        UPDATE CONFLICT HISTORY
                        IF READ/UPDATE BASED ON TEMPORARY
                        VERSION THEN
                             MARK NEW VERSION AS t(w)
                        ELSE
                             MARK NEW VERSION AS t(r)
                   ELSE
                        UPDATE CONFLICT HISTORY
                        READ/UPDATE DATA OBJECT
                        MARK NEW VERSION AS t(w)
              ELSE
                   READ/UPDATE DATA OBJECT
                   MARK NEW VERSION AS t(r)
              PUSH ENTRY ONTO DO LOG
              IF NO CONFLICT
                   RELEASE LOCK
              ENTER DETECT NON-SR EXECUTION PHASE
         ELSE
              ENTER LOCK QUEUE
              SEND MESSAGE TO OWN INITIATING SITE
              GIVING CONFLICT HISTORY AND LOCATION
              WAIT FOR LOCK TO BE RELEASED
              ENTER EXECUTION PHASE
```

```
DETECT NON-SR EXECUTION:
IF CONCURRENT SUBTRANSACTIONS THEN
    MERGE SUBTRANSACTION CONFLICT HISTORIES
IF CONFLICT HISTORY EMPTY THEN
    IF RECEIVE CONFLICT HISTORIES THEN
         SEND MESSAGE TO INITIATING SITE OF CONFLICTING
         TRANSACTION SAYING "READY TO COMMIT"
    ENTER COMMIT PHASE
ELSE
    FOR EACH TRANSACTION IN CONFLICT HISTORY DO
         SEND COPY OF CONFLICT HISTORY TO INITIATING
         SITE OF EACH TRANSACTION
    FOR EACH CONFLICT HISTORY/PRECEDENCE RELATION
    RECEIVED DO
         CONSTRUCT PRECEDENCE RELATION
         IF PRECEDENCE RELATION SHOWS NON-SR EXECUTION THEN
              RESTORE SERIALIZABLE EXECUTION
                   IF TRANSACTIONS STILL EXECUTING DO
                        SELECT LEAST COSTLY TRANSACTION
                        PAIR FROM AMONG TRANSACTIONS THAT
                        ARE STILL EXECUTING
                   ELSE
                        SELECT LEAST COSTLY TRANSACTION
                        PAIR
                   TRANSACTIONS IN SELECTED TRANSACTION
                   PAIR ROLLBACK TO SITE OF CONFLICT AND
                   EXECUTE IN OPPOSITE ORDER
              UPDATE CONFLICT HISTORY
              IF READ/UPDATE BASED ON TEMP VERSION THEN
                   MARK NEW VERSION AS t(w)
              ELSE
                   MARK NEW VERSION AS t(r)
              IF TRANSACTION COMPLETED THEN
                   ENTER COMMIT PHASE
              ELSE
                   CONTINUE WITH EXECUTION PHASE
         ELSE
              SEND PRECEDENCE RELATION TO INITIATING SITE
              OF TRANSACTION ADDED TO PRECEDENCE RELATION
    IF RECEIVE "READY TO COMMIT" MESSAGE THEN
         SEND "READY TO COMMIT" MESSAGE TO INITIATING SITE
         FROM WHICH RECEIVED MOST RECENT PRECEDENCE RELATION
         ENTER COMMIT PHASE
```

```
COMMIT PHASE:
IF ALL TEMP VERSIONS ARE t(r) THEN
    SEND COMMIT MESSAGE TO ALL SITES AT WHICH
    TRANSACTION EXECUTED
ELSE
    FOR EACH t(w) VERSION DO
        IF t(w) SITE REPORTS "ABORT" THEN
            ROLLBACK TO SITE OF ABORT AND RE-EXECUTE
            REMAINDER OF SUBTRANSACTION
            ENTER DETECT NON-SR EXECUTION PHASE
        ELSE
            IF ALL t(w) VERSIONS REPORT "READY TO COMMIT"
            THEN
                SEND COMMIT MESSAGE TO ALL SITES
                AT WHICH TRANSACTION EXECUTED
```

D. EXAMPLE

An example will demonstrate how the algorithm works. Let's consider four conflicting transactions. Each data object accessed by a transaction is assumed to be located at a different site within the distributed system. For example, data object (a) is located at Node A and data object (b) is located at Node B. (See Fig. 1).

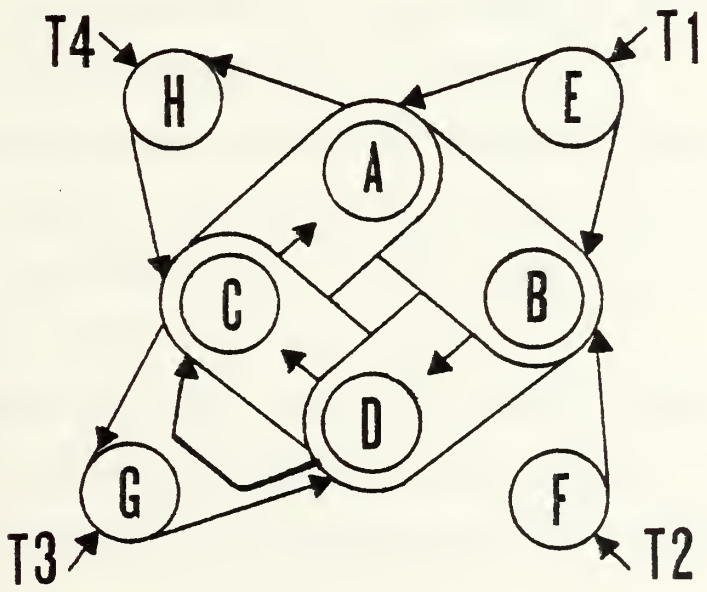

Figure 1. Cycle of Conflicting Transactions

T1 enters the system at Node E. T1 has two subtransactions. ST11 executes at Node B. ST12 executes at Node A. Upon returning to its initiating site ST11 has T2 in its conflict history. ST12 has an empty conflict history. The subtransaction conflict histories are merged to produce a transaction conflict history. The conflict history shows that T1 conflicted with T2 at data object b and the cost of the two conflicting transactions was 12. (T2T1 : 12)

T2 enters the system at Node F. It executes at Node B, at Node D and encounters a lock at Node C. The lock is held by T3. T2 waits for a time out and finds that the lock is still present. T2 sends a message to its initiating site giving its conflict history and the fact that it is blocked at Node C. T2's conflict history consists of T3 since it conflicted with T3 at data object d and the cost was 16. (T3T2 : d : 16)

T3 enters the system at Node G. It executes at Node D, at Node C and returns to Node G. Upon returning to Node G its conflict history shows that it conflicted with T4 at data object c and the cost of the two conflicting transactions was 4. (T4T3 : c : 4)

T4 enters the system at Node H. It executes at Node C, at Node A and returns to Node H. Upon returning to Node H its conflict history shows that it conflicted with T1 at data object a and the cost of the two conflicting transactions is 8. (T1T4 : a : 8)

Each transaction sends a copy of its conflict history to the initiating site of any transaction in its conflict history. T2 marks its conflict history with a "+" to indicate that it is still executing.

T1 : (T2T1 : b : 12) --> T2

T2 : (T3T2 : d : 16+) --> T3

T3 : (T4T3 : c : 4) --> T4

T4 : (T1T4 : a : 8) --> T1

Each transaction constructs a precedence relation from its conflict history and from the conflict history, which has been sent to it.

```
T1 : (T2T1 : b : 12
        T1T4 : a : 8)

T2 : (T3T2 : d : 16+
        T2T1 : b : 12)

T3 : (T4T3 : c : 4
        T3T2 : d : 16+)

T4 : (T1T4 : a : 8
        T4T3 : c : 4)
```

None of the transactions is able to commit or to detect that non-serializable execution has occurred. Therefore, each transaction sends a copy of the precedence relation, which it has just constructed, to the initiating site of the transaction, which it added to its own conflict history to produce the precedence relation.

T1 --> T4

T2 --> T1

T3 --> T2

T4 --> T3

Again each transaction constructs a new precedence relation using its own relation and the relation that it receives.

```
T1 : (T3T2 : d : 16+
        T2T1 : b : 12
          T1T4 : a : 8)

T2 : (T4T3 : c : 4
        T3T2 : d : 16+
          T2T1 : b : 12)

T3 : (T1T4 : a : 8
        T4T3 : c : 4
          T3T2 : d : 16+)

T4 : (T2T1 : b : 12
        T1T4 : a : 8
          T4T3 : c : 4)
```

None of the transactions is able to commit or to detect that non-serializable execution has occurred. Therefore, each transaction sends a copy of its precedence relation to the initiating site of the transaction that it added to the precedence relation.

```
T1 --> T3

T2 --> T4

T3 --> T1

T4 --> T2
```

Each transaction constructs a new precedence relation from the relation that it receives.

```
T1 : (T3T2 : d : 16+
        T2T1 : b : 12
          T1T4 : a : 8
            T4T3 : c : 4
              T3T2 : d : 16+)

T2 : (T4T3 : c : 4
        T3T2 : d : 16+
          T2T1 : b : 12
            T1T4 : a : 8
              T4T3 : c : 4)
```

```
T3 : (T1T4 : a : 8
        T4T3 : c : 4
          T3T2 : d : 16+
            T2T1 : b : 12
              T1T4 : a : 8)

T4 : (T2T1 : b : 12
        T1T4 : a : 8
          T4T3 : c : 4
            T3T2 : d : 16+
              T2T1 : b : 12)
```

Each transaction can now detect that a cycle has occurred since each precedence relation begins and ends with the same transaction pair. Each transaction also has complete information about the cost of each transaction pair involved in the cycle. In this case the cost information is not necessary since one transaction is still executing. Since T2 is still executing, the transaction pair is selected which will allow T2 to execute last. In this case T2T1 is the transaction pair which must be re-executed at data object b in order to break the cycle. T1 and T2 roll back to Node B. T1 executes and returns to its initiating site with an empty conflict history and commits. T2 executes, moves to Node D and executes and moves to Node C where it will be able to execute as soon as T4 commits. T4 can commit as soon as T1 commits. When T4 commits, T3 will change its DO log entry to t(r), release its lock and commit. T2 will acquire the lock, execute and mark its version as t(w) or t(r) depending on how soon T3 commits.

If none of the transactions were still executing then the least cost transaction pair would have been rolled back to break the cycle. If more than one transaction were still executing then the least cost pair, which would allow one of the still executing transactions to execute last, would be chosen to break the cycle.

Commit will take place in the following manner. Once the cycle is broken, T1 returns to its initiating site with an empty conflict history. Since T1 did not conflict at any site, all of its temporary versions are t(r). T1 broadcasts a message to all of its sites telling them to commit their DO log entries. As soon as T1 commits, T4 will release the lock that it is holding at Node A and send a message to its initiating site saying that its temporary version at Node A is ready to commit. Since this is the only t(w) version that T4 was waiting for, T4 can now broadcast a commit message to all of its sites. As soon as T4 commits, T3 will release its lock at Node C, change its temporary version to t(r) and send a message to its initiating site saying that it is ready to commit. The initiating site will broadcast the commit message to all sites. T2 is now able to execute at Node C. If T3 has committed by the time T2 executes at Node C, then T2 will mark its temporary version as t(r) and return to its initiating site. If T3 has not committed then T2 will mark its version as t(w), will hold the lock at that site and return to its initiating site. As soon as T3 commits, T2 will release the lock at Node C, change its temporary version to t(r) and send a message to its initiating site. T2 will then broadcast a commit message to all of its sites.

In order to summarize the performance of this algorithm, it is useful to look at the number of messages that were required to detect and resolve non-serializable execution and the number that were required to commit each transaction. In order to detect and resolve non-serializable execution in this example, each site was required to send three

messages. In general, the number of messages required to detect non-serializable execution is a function of the number of sites involved in the conflict. If there are n sites involved in non-serializable execution, then n/2 + 1 messages will be required to detect the non-serializable execution. This formula has been tested for scenarios with up to eleven transactions involved in a single cycle. It should hold for any number of transactions involved in a cycle.

In order to commit the following number of messages are required. Each transaction, after returning to its initiating site and resolving any non-serializable execution, must wait to hear from all k sites at which it experienced conflict. When a transaction detects conflict at a particular data object, it marks its DO log entry as t(w). A transaction cannot commit until it has heard from all k sites at which it left a t(w) version. Thus, k sites have to send k messages, which are triggered by the previous temporary versions either being committed or aborted. Once all k messages are received by the initiating site, then s additional messages are needed to commit or abort all s sites at which the transaction executed. Of course, if the previous temporary version aborts, then the transaction must roll back to the site of the abort and execute the remainder of the subtransaction from the site of the abort.

E. OTHER CONSIDERATIONS

The algorithm, as presented above, makes several assumptions, which may not be true for every application. The purpose of this section is to outline briefly certain features of the algorithm which can be modified, as required, to meet the needs of a particular application.

1. Locks

The algorithm enforces a locking scheme which is based on the detection of conflict. If a transaction detects conflict with a previous temporary version of a data object, then the conflicting transaction holds its lock when it leaves the data object. The lock is held until the preceding transaction, which generated a temporary version, either commits or aborts. Thus, when the algorithm detects a conflict, it switches from an optimistic mode to a pessimistic mode. The point at which this switch occurs is one parameter that can be adjusted to fit the needs of a particular application. That is, a transaction will hold its lock only if it detects n previous temporary transactions with which it is conflicting (n = 0,1,2,3,...). If n = 0 is chosen, then the algorithm functions in a very similar manner to distributed two-phase locking with two-phase commit. The main difference is that the first phase of two-phase commit is identical with our transaction execution phase. The higher the n that is chosen the more optimistic the algorithm and the greater the degree of concurrency that is permitted. There is also more potential damage from the domino effect.

2. Time-out

Another parameter that can be adjusted to fit a particular application is the length of the time-out. The purpose of the time-out is to reduce unnecessary message traffic by having a transaction wait when it encounters a lock. The lock may, in fact, be removed after a certain period of time. For example, the original transaction, which caused the lock to be held, may commit. If the lock is not released in a certain

amount of time, then a message needs to be sent to the initiating site of the transaction, which is blocked by the lock. The purpose of this message is to carry the transaction conflict history so that the exchange of conflict histories may begin. The transaction, which encounters the lock may be involved in a deadlock situation. Thus, the length of the time-out period will be a function of the length of time that it takes for a typical transaction to commit. If the time-out period is too long, then the blocked transaction will be wasting time. If the time-out is too short, then there will be unnecessary message traffic.

3. Site Autonomy

In the version of the algorithm presented thus far all transactions are assumed to have the same priority. In this context, site autonomy is not an important issue. Since all transactions have the same priority, it is not necessary to allow a site to unilaterally abort a transaction in order to free the resources that it is holding. It is important to note, however, that while an optimistic concurrency control mechanism does not block resources through the use of locks, it does block a transaction since it prevents the transaction from committing. In our algorithm a transaction is allowed to pre-compute its results based on temporary versions left by a previous transaction. The conflicting transaction, however, cannot commit and exit from the system until all previous versions, on which its version is based, commit.

In certain applications, transaction priority and site autonomy may be important issues. In these cases it is possible to modify the algorithm to allow a site to abort a transaction based on a priority

among transactions. Such a policy can be incorporated into the algorithm either by using a standard two-phase commit policy with its three messages or by modifying slightly the original commit policy. If the original commit policy is to be modified, no problems are created for temporary versions marked t(w) since an initiating site cannot commit until it receives a message from these versions. If a version marked t(r) is aborted, however, the possibility exists that the abort message could be sent at the same time that the version's initiating site is sending the commit message. In order to avoid this situation a site, which wishes to abort a t(r) version, must send a message to the initiating site of the original transaction saying that unless the version is committed immediately it will be aborted. This allows for the possibility that the transaction may already have sent the commit message. In other words, a site, which wishes to abort a t(r) version, must wait for a certain period of time to ensure that its abort message is not crossing a commit message from the initiating site of the original transaction. Thus, the issues of site autonomy and priority among transactions can be incorporated into the algorithm at the cost of additional message traffic or at the cost of replacing the original commit policy with a 2PC policy.

4. Metrics

In the algorithm presented above, a metric describing the amount of work involved in the atomic actions of a pair of conflicting transactions is included as a part of a transaction's conflict history. This choice of metric should be adequate for most applications. It can be improved, where necessary, through the use of additional calculations

and, possibly, an extra message. There are two improvements which are possible. First, a transaction can add the cost of redoing the entire portion of a subtransaction, from the point of conflict to the completion of the subtransaction, to the metric. Since all of the remaining atomic actions in the subtransaction are in some way dependent on the conflicting atomic action then, if the conflicting action has to be re-executed, the remaining atomic actions in the subtransaction will also have to be re-executed. Second, when a transaction receives a conflict history from another transaction, it can add the cost of redoing the remaining atomic actions in its subtransaction, from the point of conflict to the completion of the subtransaction, to the metric. If two transactions are conflicting then both improvements can be incorporated into the algorithm without the addition of any messages. If a copy of a transaction's conflict history was not sent to the initiating site of a transaction whose conflict history is received, then one additional message is required to send the updated metric back to the transaction which sent the history. If both of these actions are taken, then the resulting metric will represent the complete cost of redoing the subtransactions of the particular pair of transactions chosen to rollback and resolve nonserializable execution. These extra steps should not be necessary for most applications, but they are available if needed.

IV. COMPARISON OF ALGORITHMS

A. INTRODUCTION

This chapter compares the performance of selected algorithms under different scenarios. The algorithms to be compared are the Ceri-Owicki algorithm and the Badal-McElyea algorithm. The Schlageter algorithm is not included because of its similarity to the Ceri-Owicki algorithm. Both are based on the Kung-Robinson proposal and a detailed consideration of both algorithms would be redundant. Since Two-Phase Locking (2PL) and Two-Phase Commit (2PC) is the standard in concurrency control, an examination of this method is also made for each scenario. The four scenarios to be considered are as follows:

(1) a non-conflicting transaction

(2) two conflicting transactions which produce serializable execution

(3) two conflicting transactions which produce non-serializable execution

(4) four transactions, which are involved in a cycle of non-serializable execution

B. A NON-CONFLICTING TRANSACTION

In this scenario, transaction T1 updates data objects a, b, c, and d, which are located at four different nodes in the system. Data object a is located at Node A and so on. T1 does not conflict with any other transactions.

1. 2PL and 2PC

T1 acquires a lock on each data object and performs its update in such a way that the results can either be committed or rolled back to the original value. No lock is released until all locks have been acquired. Once all four data objects have been locked and updated, T1 begins its 2PC. A message is sent to each site at which the transaction executed asking if the lock is still held and if the site will agree to be bound by the decision of the master site either to commit or abort. If all sites agree to commit, then the master site sends a message to each site at which the transaction executed telling each site to make its temporary results permanent and to release its lock. If at least one site aborts, then the master site sends an abort message to all sites at which the transaction executed.

This method of concurrency control requires that locks be acquired and held over a certain period of time. Locks must be held at the same time. Three messages are required to commit/abort the results at each site visited by the transaction. No results are visible to other transactions until after T1 commits.

2. Ceri and Owicki

T1 executes at each site without acquiring any locks. Each site performs a validation against local and global transactions. Since T1 is non-conflicting, the transaction successfully validates at each site. Each site sends a message to the master site saying that it is ready to commit. When successful validation messages are received from all sites, the master site sends a message telling the sites to commit and make their temporary versions permanent.

This algorithm requires no locks. Two messages are required to commit the results at each site at which the transaction executes. The one message is saved, in comparison to 2PC, because each site automatically informs the master site when it is ready to commit. This means that there is loss of site autonomy. A site can no longer unilaterally abort a transaction after the transaction has informed its master site that it is ready to commit. In 2PC a site can unilaterally abort a transaction up until the time that it agrees to abide by the decision of the master site.

3. Badal and McElyea

T1 executes at each site by acquiring a local lock, updating the data object, pushing its entry onto the DO log, releasing the lock and moving on to the next data object or returning to the initiating site. T1 can execute at the four data objects in a sequential manner or it can execute concurrently at all four data objects. All temporary versions created by T1 are immediately available to other transactions. The DO log entry at each data object is marked as t(r) since the transaction is nonconflicting. Since T1 returns to its initiating site with an empty conflict history, it is ready to commit immediately. T1 sends a message to each site at which it executed telling the site to mark its DO log entry as committed.

This algorithm requires four locks but the locks do not have to be held simultaneously. Only one message is required for each site to commit its results but, again, the savings in messages is obtained at the expense of local site autonomy. A site can not unilaterally abort

a transaction once the transaction has executed at the site. The temporary versions generated by T1 are immediately available to other transactions.

4. Summary

In this case no conflict occurred. 2PL, however, acquires a lock for each data object to ensure that no conflict can occur. In the case of a non-conflicting transaction, the amount of work in acquiring these locks is small. Three messages are necessary to commit the results at each site visited by the transaction. No results are made available until after the transaction commits.

Ceri and Owicki do not require locks. No results are made available until it is certain that the transaction will commit. Only two messages are required to commit the results of the transaction at each site.

Badal and McElyea acquire a local lock at each data object to ensure mutual exclusion over that data object. The lock is released as soon as the update on that data object is complete. One message is required to commit the results of the transaction. This reduction in messages is achieved with a reduction in site autonomy. Temporary versions of all data objects are immediately available to other transactions.

C. TWO CONFLICTING TRANSACTIONS WHICH PRODUCE SERIALIZABLE EXECUTION

In this scenario, T1 executes at data objects a, b, c, and d, which are located at four different nodes in the system. T2 immediately follows T1 and executes at the same data objects in the same order.

1. 2PL and 2PC

T1 again acquires a lock at each data object and performs its update in such a way that the results can either be committed or rolled back to the original values. Once all of the locks have been acquired and all of the temporary updates made, the master site sends a message to each site asking if the lock is still held and if the site agrees to abide by the master site's decision to commit or abort. If all sites answer in the affirmative, then the master site tells all sites to commit their results and release their locks.

T2 encounters T1's lock at data object a and waits there until T1 commits its results and releases its locks. Then T2 acquires the same locks and performs its updates. Each transaction has to acquire four locks simultaneously in order to execute and commit its results. Three messages are required for each transaction to commit its results at each site where the transaction executes. The transactions are forced to execute in a serial manner. T2 follows T1. No results are available from a transaction until after it commits.

2. Ceri and Owicki

T1 executes at a, b, c, and d by creating a temporary copy of each data object and making its updates on the copies. All updates on the temporary copies are done during the transaction's read phase. The transaction successfully validates at each site. Once the master site has been informed of the successful validation at each site, it sends a message to each site telling the site to commit its results by making

the temporary copy permanent. The results of the transaction are not available until after the transaction commits.

T2 executes at a, b, c, and d shortly after T1. As long as T1 is still executing, T2 is unable to validate successfully. T2's read and write set intersects T1's write set. T2 will have to abort at each site until T1 commits its results. Thus, the performance of this algorithm can not possibly be any better than 2PL and, under certain circumstances, it can be considerably worse. For instance, suppose that T2 begins shortly before T1 commits its results. T2 will execute in its entirety, abort because its read set intersects the write set of T1, and then execute again in its entirety. This is true since transactions do not make their results available to other transactions until they commit. Thus, if another transaction reads a data object, which is in the process of being updated by another transaction, then the transaction, which read the old value, will have to abort. When the original transaction commits, however, there is no way to notify other transactions, which have read incorrect values, that they must abort. The transactions must run in their entirety and then abort.

There are no locks required to execute this algorithm. A variable number of messages is required in order for each transaction to commit its results. T1 requires two messages for each site at which it executes. T2 requires four or more messages to commit its results at each site since the transaction has to abort and restart at least once before it can successfully validate and commit. The degree of concurrency for the algorithm is certainly no better than 2PL and, in some cases, considerably worse.

3. Badal and McElyea

T1 can move sequentially from site to site or it can execute concurrently at all sites. At each site T1 acquires a local lock, reads and updates the data object, pushes its entry onto the DO log, releases its lock and either moves to the next site or returns to the initiating site. Since T1 does not conflict with any other transactions, it returns to its initiating site with an empty conflict history. T1 sends a message to each site at which it executed telling the site to mark its DO log entry as committed. T2 can likewise execute at the four sites in a sequential or concurrent manner. At each site T2 detects that it is conflicting with T1 but since the conflict history that T1 left at each site is empty, then T2 does not detect non-serializable execution. When T2 returns to its initiating site its conflict history contains T1. Therefore T2 sends a copy of its conflict history to T1. T1 responds that its conflict history is empty and, thus, no non-serializable execution could have occurred between them. Since T2 conflicted with T1 at each data object, it marked its DO log entries as t(w) and held its locks. As T1 commits at each data object, T2's log entry is changed to t(r), the lock is released and T2's initiating site is notified that its version of the data object at a given site is ready to commit. Once T2's initiating site has been informed that all of its versions are ready to commit, it sends a message to each site telling the site to mark its version as committed in the DO log. It should be noted that no other transactions are permitted to access the data objects once a conflict is detected. Each time that T2 detects a conflict, it holds the lock on that data object. This prevents

the effects of any potential non-serializable execution from spreading throughout the system.

There are two messages required to detect that only serializable execution has occurred. T1 needs one message to commit its results at each site because it did not detect any conflict. T2 requires two messages since it detected conflict at each site. The degree of concurrency in this algorithm is relatively high. Both transactions are able to execute concurrently, In each case, T2 gets the updated version left by T1 at each data object. Thus, when T1 commits, T2 can commit because, in a sense, it has just pre-computed its results based on the assumption that T1 will eventually commit. If T1 had aborted, then T2 would have also had to abort because its pre-computed results would have been incorrect. The fact that T2 detected a conflict with T1 and held its lock at each data object where conflict was detected prevented any potential problems from affecting other transactions in the system. The locks also sequestered the resources that would be needed if the conflict turned out to be non-serializable execution.

4. Summary

In this case, 2PL and the Ceri-Owicki algorithm prevent concurrent execution and force the transactions to execute in a serial manner. In 2PL two transactions require 6s messages, where s is the number of sites a transaction accessed. In the Ceri-Owicki algorithm at least 6s messages are required. Using the Badal-McElyea algorithm the transactions are allowed to execute concurrently and 3s messages are required.

D. TWO CONFLICTING TRANSACTIONS WHICH PRODUCE NON-SERIALIZABLE EXECUTION

In this scenario, T1 wants to execute at data objects a, b, c, and d, which are located at four different sites throughout the system. T1 begins executing at the site where a is located. It moves to the other sites in a sequential or concurrent manner. T2 wants to execute at data objects d, c, b, and a. T2 begins executing at the site where d is located. It moves to the other sites in a sequential or concurrent manner. This scenario will produce non-serializable execution. At data object a the transaction will execute in the order T1T2. At data object d the transactions will execute in the order T2T1. At data objects b and c the order of the transactions will depend on which one gets to the object first.

1. 2PL and 2PC

T1 begins acquiring locks at data object a. At some time T1 will attempt to lock a data object which is already locked by T2. T2 will encounter the same situation in reverse by starting to acquire locks at data object d. This results in a deadlock situation. T1 holds a lock needed by T2 and T2 holds a lock needed by T1. After a certain period of wait, a deadlock detection mechanism is invoked to detect and resolve the deadlock. Either T1 or T2 will have to abort and give up its locks so that the other transaction can run to completion. As soon as that transaction commits its results and releases its locks the other transaction can acquire the locks and run to completion. The resulting execution will be somewhat slower than if both transactions had executed

in a serial manner. The locks and deadlock detection mechanism represent overhead in terms of time and resources that is not present in the serial case.

2. Ceri and Owicki

This algorithm allows T1 and T2 to run to completion at each data object. Both transactions, however, will fail to validate successfully. At Node A, T2 will have to abort because its read set intersects T1's write set. At Node D T1 will have to abort because its read set intersects with T2's write set. Thus, both transactions will have to abort and restart. If no mechanism is used to ensure that one or the other of the transactions starts at a different time, then the same results may continue to be repeated indefinitely. That is, the algorithm recognizes that conflict has occurred and it forces the transactions to abort but it does not prevent the same situation from developing on the restart. The reason for this problem appears to be that the concurrency control mechanism fails to distinguish between conflict and non-serializable execution. As soon as the algorithm detects conflict the transactions, which have detected the conflict, are forced to abort and restart. The problem is that this policy may not be successful in the case of non-serializable execution. Merely restarting the transaction will not necessarily eliminate the non-serializable execution. It seems that some mechanism should be introduced to stagger the restart so that one transaction can be allowed to run to completion followed by the other. In any case the transactions will not run as well as if they had executed serially. The transactions will take at least twice as long to

run even considering that the transactions only abort and restart one time. The cost of running the transactions, then aborting and restarting them, all represents overhead that is not present if the transactions execute serially.

3. Badal and McElyea

In this scenario the transactions can execute either sequentially or concurrently. The sequential case will be described but the detection of the non-serializable execution is handled in the same way if the transactions execute concurrently at the data objects.

T1 begins executing at Node A and T2 begins executing at Node D. Each transaction acquires a local lock, performs the update on the data object, pushes its entry onto the DO log, and releases the local lock. Neither transaction initially detects any conflict. At some point, however, the transactions will overlap and detect conflict with each other. The other transaction's conflict history, which was left as part of its DO log entry, will be empty, so the transaction detecting the conflict will merely add the transaction to its conflict history and continue on to the next data object. The lock, which was acquired by the transaction which detects a conflict at a data object, will not be released. Upon returning to their initiating sites, each transaction contains the other in its conflict history. Each transaction sends the other a copy of its conflict history. Each conflict history will contain a metric to indicate the cost of redoing each transaction pair where a conflict was detected. Upon receiving the conflict history from the other transaction, it will be immediately apparent that non-serializable execution

has taken place. Since a metric on the cost of each conflicting pair of transactions is included with the conflict history, it will also be immediately apparent to each transaction how the conflict can be resolved in the most economical manner. Depending on the exact timing of the transactions, the two transactions will have to rollback to one or, more probably two, data objects and reverse the order of execution. Once the non-serializable execution has been resolved, one transaction will have an empty conflict history and can commit immediately upon returning to its initiating site. The other transaction can commit as soon as the first transaction commits. It should be noted that the locking scheme sequesters all of the resources that are needed to resolve the non-serializable execution. No other transactions are permitted to access the resources that are in question.

The messages, which are required to detect the non-serializable execution, and the rollback of portions of each transaction all represent overhead which is not present in the serial execution of the two transactions. This cost, however, is partially offset by the fact that the two transactions are allowed to execute concurrently. In addition, not all of the two transactions have to be redone. Portions of each transaction are rolled back and redone but these portions are selected on the basis of being the most economical.

4. Summary

In this particular scenario none of the algorithms performed as efficiently as would a serial ordering of the two transactions. Once the non-serializable execution of the transactions was detected under 2PL,

the transactions were forced to execute serially. The work that was done prior to the detection of deadlock was somewhat wasted effort since some of it had to be aborted. Time and resources were also required to detect the deadlock. Under the Ceri-Owicki algorithm the nature of the problem went undetected. The conflict was detected and aborted but nothing was done to prevent the recurrence of the same situation as soon as the transactions restarted. With the Badal-McElyea algorithm both transactions were permitted to execute concurrently. Once execution had completed an exchange of conflict histories revealed non-serializable execution. The most economical means of restoring serializable execution was chosen from information provided in the exchange of conflict histories, but, as in the case of 2PL, both transactions are executed serially.

E. FOUR TRANSACTIONS INVOLVED IN A CYCLE

The scenario used for this comparison is the same as the one used to illustrate the workings of the Badal-McElyea algorithm in Chapter 3. There are four transactions, T1, T2, T3, and T4, which are attempting to execute concurrently. See Figure 1 and Chapter 3 for a detailed description of the scenario.

1. 2PL and 2PC

T1 acquires locks at Nodes E and A but is not able to acquire a lock at Node B because T2 holds the lock on data object b. T2 acquires locks at Nodes F and B but is unable to acquire locks at D and C. The lock at D is held by T3 and the lock at C is held by T4. T3 acquires a lock at Nodes G and D but is not able to acquire a lock at C because T4 already holds it. T4 acquires a lock at Nodes H and C but is unable to acquire a lock at A because T1 holds it.

This situation is a deadlock. Each transaction is prevented from acquiring the locks that it needs to complete execution. After a certain wait period, the deadlock mechanism is invoked to resolve the conflict. The cycle is broken by aborting one or more of the transactions which are involved in the cycle. Once a transaction is aborted, then another transaction can acquire the lock that it needs to complete its execution and commit. Once a transaction has committed and released its locks, another transaction can acquire a needed lock and so on. In effect, the transactions will end up executing in a serial manner.

2. Ceri and Owicki

This algorithm permits all of the transactions to execute concurrently. None of the transactions is able to validate successfully. T1 fails at Node B because its read set intersects T2's write set. T2 is unable to validate because T3's write set intersects its read set at Node D and both T4's and T3's write sets intersect its read set at Node C. T3 is unable to validate because the write set of T4 intersects its read set at Node C. T4 is unable to validate because T1's write set intersects its read set at Node A. This is another example of non-serializable execution. Aborting and restarting the transactions involved in the non-serializable execution does not do anything to eliminate the non-serializable execution. The same execution may be repeated as soon as the transactions restart. In order to prevent the same cycle from repeating itself, some mechanism needs to be introduced to vary the restart of the transactions. The overhead for this algorithm is in the form of aborting and restarting transactions.

3. Badal and McElyea

The detailed description of this algorithm is found in Chapter 3. In summary, all four transactions execute concurrently. T1, T3, and T4 complete their execution and return to their initiating sites. T2 encounters a lock at Node C. After waiting for a time-out, T2 finds the lock still present and sends its conflict history to its initiating site. Conflict histories are exchanged and the non-serializable execution is detected. Each transaction involved in the cycle sends three messages in order to detect the non-serializable execution. In this scenario, $n = 4$ and $n/2 + 1 = 3$. Once the cycle is detected, it is resolved so that T2 can execute last. This is done since T2 is still executing and such a policy will prevent non-serializable execution from occurring among the transactions involved in the cycle once T2 resumes its execution. The cycle is broken at Node B. T1 and T2 have to rollback to Node B and re-execute. T2 might also have to re-execute at Node D since the action that T2 took at Node D was in some way dependent on the results that it obtained at Node B.

Once the cycle is broken and serializable execution is restored, then transactions are able to commit. T1 has an empty conflict history and is able to commit immediately. Once T1 commits, T4 can commit. When T4 commits, T3 will release its lock at Node C and T2 can complete its execution. In the meantime T3 will commit and T2 can commit.

The overhead for the algorithm in this case is somewhat similar to 2PL but fewer messages are required. In 2PL, three messages are required for each site where the transaction executed. In the

Badal-McElyea algorithm, a message is required from each site where conflict occurred and then one message is sent to each site where the transaction executed.

4. Summary

In this case, the 2PL and Badal-McElyea algorithm seem to function in a similar manner. 2PL experiences a deadlock, resolves the deadlock and allows the transactions to continue in a serial manner. The Badal-McElyea algorithm allows the transactions to run to completion or, in one case, to deadlock. The non-serializable execution is resolved and the transactions either commit or continue execution. The 2PC requires more messages to commit its results but the fewer messages in the Badal-McElyea algorithm result in a loss of site autonomy for a longer period of time. The degree of concurrency in the Badal-McElyea algorithm is higher than 2PL but a certain portion of the transactions must be repeated in order to resolve the non-serializable execution. 2PL also requires portions of some transactions to be aborted and repeated in order to resolve deadlock.

The Ceri-Owicki algorithm seems to perform poorly in this case. It seems that the time required for the transactions to execute is worse than if the transactions just executed serially. It is not altogether clear that the transactions would ever be able to commit. The problem seems to lie in the fact that the algorithm merely notes conflict and forces transactions experiencing conflict to abort and restart. No attempt is made to recognize the nature of the conflict and take corrective action. When the conflict in fact is non-serializable execution, there is no effort made to keep the non-serializable execution from repeating in the same manner once the transactions restart.

V. CONCLUSIONS

The Badal-McElyea algorithm presented here appears to perform better than the other concurrency control mechanisms for the following reasons. When there is no conflict, it produces minimum overhead. There are fewer messages required to commit a transaction's versions, under the Badal-McElyea algorithm, but this reduction is achieved at the loss of site autonomy for a longer period of time. In 2PC a site can unilaterally abort a transaction at any time until it agrees to abide by the decision of the master site. Under the Badal-McElyea algorithm a site cannot abort a transaction after the transaction has left the site. A solution to this loss of site autonomy can be achieved by sending an additional message. The resulting algorithm still uses fewer messages than 2PC but the differences are not so significant.

The Badal-McElyea algorithm achieves some increase in concurrency over 2PL and 2PC by allowing a transaction to access results generated by other transactions, which are not yet committed. This, in fact, seems to be optimistic in the sense that once a transaction has executed on a data object, it is assumed that the transaction will commit. In those cases where a transaction doesn't commit, then certain results will have to be undone. The amount of work that will have to be undone, however, can be strictly limited by a locking mechanism, which prevents further access to a data object once the level of conflict reaches a certain point. This locking mechanism automatically sequesters the resources that will be needed to resolve any non-serializable execution.

The Ceri-Owicki algorithm, on the other hand, calls itself optimistic but, in fact, does not seem optimistic at all. Once a transaction accesses a data object, then it effectively locks that data object. In fact the results are worse than if the object were locked. Transactions are still allowed to access the object but if the original transaction commits, then all of the transactions which have accessed the data object are forced to abort. There is no mechanism to immediately abort such transactions once the original transaction commits. Any other transactions which access the data object just run to completion and then are forced to abort and restart. This can very well take longer than if the transaction were blocked by a lock at the data object and ready to resume execution as soon as the lock is released. This policy hardly seems optimistic. The assumption seems to be that the original transaction will abort. If this happens, then a transaction which accessed the same data object can commit. But if the original transaction commits, then any transaction which accessed the same data objects must abort. This policy does not really seem to merit the title of optimistic.

Another shortcoming of the Ceri-Owicki algorithm is its performance when it encounters non-serializable execution. The original Kung-Robinson proposal points out that the cost of the deadlock detection and resolution mechanism must be added to the cost of the locking mechanism, when calculating overhead, since none of these items would be necessary if all transactions executed in a serial manner. The optimistic algorithms, which are based on the Kung-Robinson proposal, do away with locks and, thus, eliminate the possibility of deadlock. What they can't do

away with is the possibility of non-serializable execution. But they don't have any policy or mechanism to deal with non-serializable execution once it occurs. Aborting the transactions does not ensure that the non-serializable execution won't be repeated as soon as the transactions are restarted. 2PL detects non-serializable execution through the use of a deadlock detection mechanism. The non-serializable execution is resolved and the transactions are forced to execute serially. The Badal-McElyea algorithm detects non-serializable execution through the exchange of conflict histories and the construction of precedence relations. The non-serializable execution is resolved by rolling back certain portions of the transactions involved in the non-serializable execution. The Ceri-Owicki algorithm merely detects that some kind of conflict has occurred and the transactions are forced to abort and restart. Nothing in the algorithm is designed to prevent the non-serializable execution from repeating in exactly the same manner.

LIST OF REFERENCES

1. Flynn, J., et. al., "Operating Systems, An Advance Course," Lecture Notes in Computer Science 60, Springer-Verlag, 1980, p. 430.

2. Ullman, J. D., Principles of Database Systems, Computer Science Press, 1980, p. 329.

3. Kung, H. T. and Robinson, J. T., "On Optimistic Methods for Concurrency Control," ACM Transactions on Database Systems, v. 6, No. 2, June 1981.

4. Ceri, S. and Owicki, S., "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases," Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, Asilomar, February 1982.

5. Schlageter, G., "Optimistic Methods for Concurrency Control in Distributed Database Systems," Proceedings Seventh International Conference on Very Large Databases, Cannes, France, September 1981.

6. Badal, D. Z., "Concurrency Control Overhead or Closer Look at Blocking vs. Nonblocking Concurrency Control Mechanisms," Proceedings of the Fifth Berkeley Conference on Distributed Data Management and Computer Networks, San Francisco, February 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93940	2
4.	D. Z. Badal, Code 52Zd Department of Computer Science Naval Postgraduate School Monterey, CA 93940	3
5.	Major William P. McElyea 2622 Culpeper Road Alexandria, VA 22308	5
6.	Commander Geir Jevne Royal Norwegian Navy SMC 1675 Naval Postgraduate School Monterey, CA 93940	1
7.	Captain Peter Jones MCCDPA MCDEC Quantico, VA 22134	1

Thesis
M1843
c.1
198799
McElyea
Optimistic concur-
rency control for
distributed databases.